OUTRIDER: Optimizing the mUtation Testing pRocess In Distributed EnviRonments

Pablo C. Cañizares[1], Alberto Núñez[1], and Juan de Lara[2]

[1] Universidad Complutense de Madrid, Madrid, Spain.
[2] Universidad Autónoma de Madrid, Madrid, Spain.

Abstract

The adoption of commodity clusters has been widely extended due to its cost-effectiveness and the evolution of networks. These systems can be used to reduce the long execution time of applications that require a vast amount of computational resources, and especially of those techniques that are usually deployed in centralized environments, like testing. Currently, one the main challenges in testing is to obtain an appropriate test suite. Mutation testing is a widely used technique aimed to generate high quality test suites. However, this technique requires a high computational cost to be executed.

In this work we propose an HPC-based optimization that contributes to bridging the gap between the high computational cost of mutation testing and the parallel infrastructures of HPC systems aimed to speed-up the execution of computational applications. This optimization is based on our previous work called `EMINENT`, an algorithm focused on parallelizing the mutation testing process using MPI. However, since `EMINENT` does not efficiently exploits the computational resources in HPC systems, we propose 4 different strategies to alleviate this issue. Moreover, a thorough experimental study has been carried out by using different applications to analyze the scalability and performance obtained with `OUTRIDER`.

Keywords: Parallel and Distributed Computing, High Performance Computing, Mutation testing

1 Introduction

Wired communications have experienced a notorious growth with the use of fiber optic, reaching in experimental environments a speed of 560 Gbit/s [10]. The main features of these networks, such as low latency and very high bandwidth, have made commodity clusters a cost-effective solution, it being the main source for high performance computing (in short, HPC). As an example, in the most recent survey of the fastest 500 computers in the world [12], 86.4% are clusters. This trend has allowed several research advances in scientific computing by using HPC techniques [16, 1]. Moreover, techniques that require a long execution time and are usually executed in centralized environments, like testing, can be deployed in clusters in order to reduce its high computational cost [2].

At present, testing is one of the most extended mechanisms to check the correctness of software [13]. However, in order to properly check the validity of a program, it is required to generate an appropriate test suite (in short, TS), which in most cases is a difficult and challenging task. Moreover, a large TS requires a very long execution time. Fortunately, there exist mechanisms, like mutation testing (in short, MT), focusing on improving the design of high quality TS. Basically, MT is based on applying mutation operators to programs that make small syntactic changes in order to produce a set of mutants. The idea is that if a TS is able to distinguish between a program and the generated mutants, it should be good at detecting a faulty implementation. Thus, the effectiveness of a TS is established on the basis of calculating the number of mutants that are distinguished from the original program.

In recent years, MT has been successfully applied in several fields [8, 4]. However, MT is computationally expensive because a large TS is executed over a vast collection of mutants. It is therefore required high computational resources in order to speed-up the testing process.

In this paper we present `OUTRIDER`, an HPC-based optimization to improve the overall performance of the MT process. Our approach uses the `EMINENT` algorithm as basis [2], which focuses on reducing the execution time of MT by parallelizing the testing process in HPC systems. However, since `EMINENT` does not properly exploit the resource usage in HPC systems, we use our proposed optimization consisting of 4 different strategies to alleviate this issue. These strategies are summarized as follows:

- Parallelizing the execution of the TS over the original application. While existing works in the literature sequentially execute the TS over the original application, we propose to distribute the test cases among different processes to be executed in parallel.
- Sorting the TS by using the execution time of each test case as sorting criteria. Since the TS is executed over the original application in the initial phase of the testing process, the time required to execute each test case can be collected. In the next phase of the testing process, where the TS is executed over each mutant, this information can be used to specify the order of execution for each test case. Hence, the test case requiring the shortest amount of time to be executed is processed in the first place. In general, for each test case, the shorter execution time is required, the greater priority to be executed.
- Enhancing the test case distribution. This strategy focuses on maximizing the number of different mutants executed in parallel. We say that a mutant is executed in parallel when different test cases are executed over this mutant, in different processes, at the same time. In this case, the resource usage efficiency may decrease if the test case that kills the mutant is executed in parallel with other test cases. Consequently, the execution of those test cases that do not kill the mutant is useless for obtaining the final results and, therefore, the computational resources are not efficiently used.
- Categorizing cloned and equivalent mutants. This strategy is based on grouping those mutants that are clones and equivalents. We say that two different mutants are clones when the resulting executable files obtained from their compilation are identical. A mutant is considered equivalent with respect to the original application when there is no test case that kills this mutant. The main goal of this strategy consists in avoiding the complete execution of both equivalent and cloned mutants.

The remainder of this paper is organized as follows. Section 2 presents the state of the art. Section 3 describes `OUTRIDER` in detail. Next, in Section 4, we present some performance experiments by analyzing the suitability of the proposed optimization. Finally, in Section 5, we present our conclusions and future work.

2 State of the art

Since the first contributions in MT, there has been a constant effort to alleviate its high computational cost. As a result, different works aimed to improve the performance of the MT process can be found in the literature. These contributions can be classified in two main groups.

The first group focuses on reducing the total number of mutants without losing a significant effectiveness. Among them, it is specially relevant *mutant sampling*, a technique based on randomly selecting a subset of mutants [22], *mutant clustering*, which selects a collection of mutants by using clustering algorithms [11] and *high order mutation*, that generates a reduced set of mutants, which are created by applying multiple mutation operators [23].

The second group consists of those contributions focusing to reduce the execution time of the MT process. In this field, there are different proposals based on shared-memory, which can be divided in two different categories: Single Instruction Multiple Data systems [5, 9] and Multiple Instruction Multiple Data systems [7, 21]. The main issue of this kind of systems is the lack of scalability in the number of processors and in the memory system.

Moreover, there exist multiple contributions based on distributed memory [3, 20]. It is worth to mention the contribution of Mateo and Usaola. They presented a dynamic distribution algorithm, called PEDRO (Parallel Execution with Dynamic Ranking and Ordering), that uses Factoring Self-Schedulling ideas [19]. Also, they introduce *Bacterio*^{P}, a parallel extension of the MT tool *Bacterio* [18], that uses *Java-RMI* [6] in order to communicate processes through the network. The results obtained in this proposal are better than those obtained in previous works. However, although the performance and the scalability achieved in this contribution is better than those obtained in shared-memory approaches, the used communication mechanism acts as a system bottleneck and, consequently, it limits the overall system performance [17].

In our previous work we proposed `EMINENT` [2] , a dynamic distributed algorithm focused on HPC systems and designed to reduce the high computational cost of MT. In `EMINENT` we use MPI to interchange information between processes, which alleviates the previously described bottleneck issue [17]. However, we consider that the distribution of the workload can be improved in order to increase both the resource usage efficiency and the level of parallelism.

3 Description of `OUTRIDER`

In this paper we propose an optimization that contributes to bridging the gap between one of the main limitations of MT, its high computational cost, and the main advantages provided by HPC systems, parallel infrastructures to speed-up the execution of computational applications. In this section we present in detail our proposal, called `OUTRIDER`, which consists of 4 different strategies to improve the overall performance of the MT process.

3.1 Parallelizing the execution of the TS over the original application

Usually, the sequential execution of a TS over the original program is an issue that hampers the scalability of the MT process [19]. This phase of the testing process becomes specially relevant in those cases where the application under test requires a long execution time and when the TS consists of a large number of test cases. Consequently, the scalability of the system is compromised due to the lack of parallelism, which is generally reflected in a low system performance.

In order to alleviate this issue, we propose to exploit the resources of the system by executing the TS over the original program in parallel. Basically, this strategy consists in distributing the execution of each test over the original program among different processes, which are executed in the available CPU cores of the system.

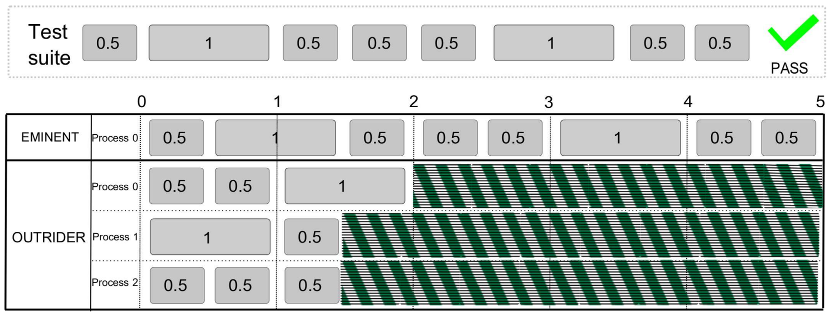


Figure 1: Execution of a TS over the original program using `EMINENT` and `OUTRIDER`

Figure 1 illustrates the comparison between the execution of a TS over the original program using `EMINENT` and `OUTRIDER`. The schema at the top of the figure shows the sequential execution of the TS, where each rectangle represents a test case and the number inside it shows the required slots of time to be executed. This TS consists of 8 test cases and has a total duration of 5 slots of time. While the TS is sequentially executed in one processor using `EMINENT`, `OUTRIDER` parallelizes the execution of the TS using 3 different processes. Also, this example shows that the distribution of the TS improves the overall performance, obtaining a speed-up of 2.5.

3.2 Sorting the TS

In MT, test cases are executed over a mutant until one of these situations occurs: the mutant is killed or the TS is completely executed and the mutant is kept alive. It is therefore desired that the first test case to be executed kills the mutant, it being the ideal situation. Unfortunately, we cannot determine those test cases that kill the mutant before executing them. However, we can use relevant information gathered from the execution of the TS over the original program, like the execution time of each test case.

This strategy uses this information to specify the execution order of the test cases. Thus, test cases are sorted by using its execution time as sorting criteria. As a result, the fastest test case is processed in the first place, while the slowest test case is executed last. The idea is to minimize the required time to kill a mutant. Although sorting the TS has a computational cost, we suppose that applying this strategy would reduce the overall execution time.

TC$_{EMI}$	**TC**$_{OUT}$	**ExecTime**$_{EMI}$	**ExecTime**$_{OUT}$	**Acc**$_{EMI}$	**Acc**$_{OUT}$	Improv.
1	2	1175	139	1175	139	-848
2	5	139	211	1314	350	1175
3	3	498	498	1812	848	964
4	1	1471	1175	3283	2023	-211
5	4	211	1471	3494	3494	3144

Table 1: Execution time, in seconds, of 5 test cases over a mutant using `EMINENT` and `OUTRIDER`

In order to illustrate the concepts of this strategy, we present a running example. Table 1 shows the execution of a TS consisting of 5 test cases over a mutant. The first two columns, TC$_{EMI}$ and TC$_{OUT}$, refer to the order of execution for each test case using `EMINENT` and

OUTRIDER, respectively. The next two columns, ExecTime$_{EMI}$ and ExecTime$_{OUT}$, represent the execution time for each single test case. These are followed by the two columns that represent the accumulative time required to execute the test cases using EMINENT and OUTRIDER. The last column shows the improvement obtained by comparing OUTRIDER and EMINENT, where positive values indicate that OUTRIDER executes faster than EMINENT and negative values show otherwise. This improvement is calculated by taking into account that there is a test case that kills the mutant, which is indicated in the first column. In this example, OUTRIDER obtains better results than EMINENT when the test case 2, 3 or 5 kills the mutant, obtaining an improvement in the total execution time of 1175, 964 and 3144 seconds, respectively. On the contrary, EMINENT executes faster than OUTRIDER when test case 1 or 4 kills the mutant.

3.3 Enhancing the test case distribution strategy

In order to increase both the level of parallelism and the resource usage efficiency, we propose a strategy that improves the workload distribution presented in EMINENT, which additionally considers the number of remaining mutants to be completely executed, the number of processes involved in the testing process and the number of processes that are executing each mutant. The idea is to improve the resource usage efficiency by maximizing the number of different mutants being executed in parallel. Thus, when the number of remaining mutants to be executed is greater or equal than the number of available processes, each single process executes a different mutant. On the contrary, the remaining mutants to be completely processed are proportionally distributed among the available processes.

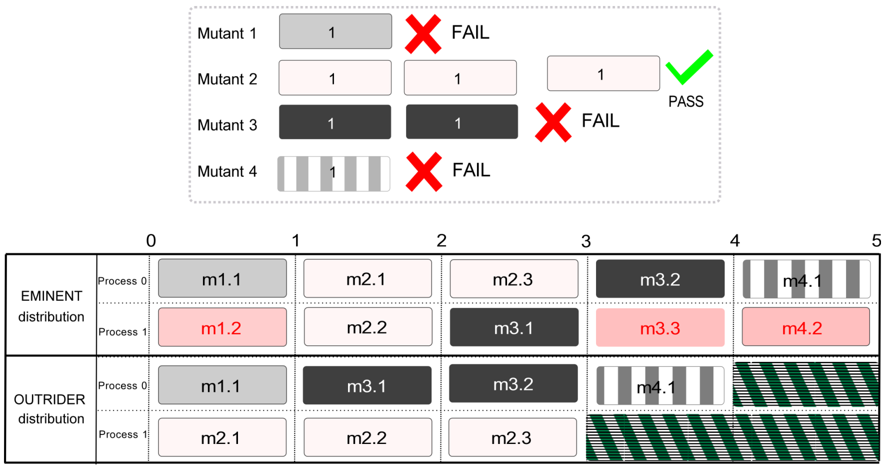


Figure 2: Workload distribution using EMINENT and OUTRIDER

Figure 2 illustrates a running example to compare two different distribution strategies. In this example a TS consisting of 3 test cases is executed over 4 mutants using 2 processes, each one having a dedicated CPU core. The schema at the top of Figure 2 shows the execution of the TS over each mutant, where test case 1 kills mutant 1 and 4, test case 2 kills mutant 3, and mutant 2 remains alive after the execution of the TS.

The schema at the bottom of Figure 2 shows two different strategies to distribute the workload in the MT process, that is, the workload distribution presented in EMINENT and the workload distribution used in OUTRIDER. For the sake of clarity, we denote by mX.Y the execution of the test case Y over the mutant X. In this scenario, executions m1.1, m3.2 and m4.1 kill the processed mutant.

The distribution strategy used in `EMINENT` shows that the execution of some test cases is useless. For instance, m1.1 and m1.2 are executed in parallel. Although the former execution kills mutant 1, process 1 is wasting computational resources by executing m1.2, which is not necessary to kill the mutant. Since the strategy used in `OUTRIDER` maximize the number of different mutants executed in parallel, this situation is avoided in the major part of scenarios. In this example, `OUTRIDER` obtains an improvement of 20% in the total execution time.

3.4 Categorizing equivalent mutants using TCE

In MT, a mutant is considered equivalent when none of the test cases are able to kill it. The equivalence problem is one of the principal obstacles of the practical use of MT. Although it is well known that finding the equivalence between two programs is a non-decidable problem [14], there exist several heuristics that aid to find some pattern to identify this kind of mutants. In this case, due to its simplicity and its computational efficiency, we have selected the trivial compiler equivalence technique (in short, TCE), to detect both equivalent and cloned mutants [15]. This technique uses compiler optimizations in order to detect some patterns that aids to identify the equivalence between programs using a black-box scheme.

These concepts are used in our proposed strategy to detect two kinds of mutants. On the one hand, those mutants that are equivalents to the original program, which are known are *equivalents*. On the other hand, those mutants that differ from the original program but are identical to other mutants, are called *cloned* mutants.

We apply this technique after the compilation phase, where both equivalent and cloned mutants are detected. Those mutants identified as equivalents are discarded and none of them are executed. On the contrary, cloned mutants are grouped in domains, where a single mutant is known as *representative* of the domain.

During the testing phase, only those mutants that do not belong to a domain are executed, which are handled as usual. Next, for each domain, only representative mutants are processed. Once the execution of a representative mutant ends, if the mutant is killed, only the killer test is applied to the rest of the mutants of the domain, which substantially reduce the number of test case executions. On the contrary, if the representative mutant is kept alive, the rest of the mutants of the domain are managed as usual.

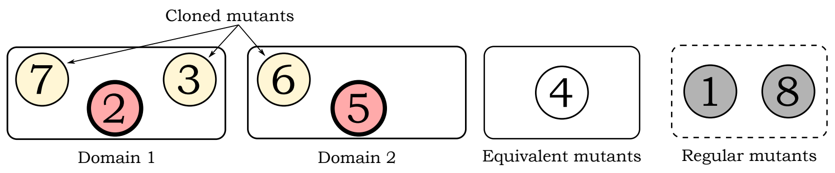


Figure 3: Categorization of cloned and equivalent mutants

In order to show the applicability of this strategy we present a running example. Figure 3 shows the execution of a MT process consisting of 8 mutants. We applied our strategy to categorize these mutants. As a result, we obtain two different domains, where Domain 1 consists of the mutant 2, 3 and 7, and Domain 2 consists of the mutant 5 and 6. Mutants with a bold border are the representatives of its domain, that is, mutant 2 is the representative mutant for Domain 1 and mutant 5 is the one for Domain 2. Mutant 4 has been detected as equivalent mutant, while mutant 1 and 8 are categorized as regular mutants.

Mutant ID	**Exec.Time**	**Time$_{EMI}$**	**Killer test time**	**Time$_{OUT}$**
1	432	432	-	432
2	245	677	-	677
3	245	922	46	723
4	456	1378	-	723
5	532	1910	-	1255
6	532	2442	164	1419
7	245	2687	46	1465
8	591	**3278**	-	**2056**

Table 2: Execution of 8 mutants using `EMINENT` and `OUTRIDER` with TCE

Table 2 shows the execution time of the MT process presented in the running example. The first two columns, *Mutant ID* and *Exec.Time*, represent the mutant ID and its execution time, respectively, $Time_{EMI}$ refers to the accumulated time when the testing process is executed using `EMINENT`. The next column refers to the execution time of the test that kills the mutant, which is calculated from the representative mutant of each domain. Finally, $Time_{OUT}$ refers to the accumulated time when the testing process is executed using `OUTRIDER`.

These results show that `OUTRIDER` executes 37% faster than `EMINENT`, that is, while `EMINENT` requires 3278 seconds to completely execute the testing process, `OUTRIDER` requires 2056 seconds. This improvement of performance is obtained because `OUTRIDER` executes less test cases than `EMINENT`. In this case, mutant 3, 6 and 7 are not completely executed because only the test case that kills them is executed instead.

4 Experiments

This section presents a thorough experimental study to analyze the scalability and performance of `OUTRIDER`. The mutant set used in these experiments has been created using the mutation framework Milu [7]. We use two different applications in the MT process. First, an image filtering application consisting of 3 algorithms to filter BMP images. Initially, all the images are located in a remote repository, which has a total size of 2,5 GB. In order to check this application, a TS consisting of 3200 test cases are executed over 250 mutants. The second application performs the multiplication of two large matrices. In this case, a TS consisting of 2000 test cases are executed over 100 mutants.

These experiments have been performed in a cluster that consists of 8 nodes interconnected through a Gigabit Ethernet network. Each node contains a Dual-Core Intel(R) Core(R) i5-3470 CPU at 3.4 Ghz with hyper-threading, 8 GB of RAM and 500GB HDD.

4.1 Performance evaluation of each single strategy in `OUTRIDER`

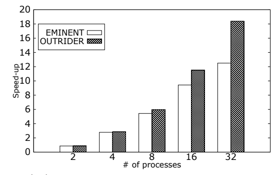
In this section, each proposed strategy in `OUTRIDER` is individually analyzed. For the sake of simplicity, we use the following notation: `S1` refers to the strategy that parallelizes the TS execution over the original program (see Section 3.1), `S2` refers to the strategy that sorts the TS (see Section 3.2), `S3` is the strategy that enhances the test case distribution (see Section 3.3) and `S4` is the strategy that categorizes both cloned and equivalent mutants (See Section 3.4).

Figure 4 shows the overall speed-up in `EMINENT` and `OUTRIDER`, with respect to a sequential execution, using 2, 4, 8, 16 and 32 processes. Each process is always executed in a dedicated CPU core. In these charts, X-axis represents the number of processes and Y-axis represents the speed-up. The first row of charts in Figure 4 shows the results obtained using the image filtering application, while the second row refers to the results obtained for checking the CPU-intensive application. Each chart analyzes a different strategy of `OUTRIDER`, that is, charts 4(a) and 4(e) analyze `S1`, charts 4(b) and 4(f) evaluate `S2`, charts 4(c) and 4(g) analyze `S3` and charts 4(d) and 4(h) evaluate `S4`.

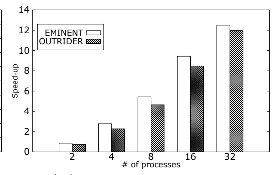
In general terms, `OUTRIDER` outperforms `EMINENT` in the major part of the evaluated scenarios. There is only one scenario where `EMINENT` executes faster than `OUTRIDER`, that is, testing the filtering application using the strategy `S2` in `OUTRIDER` (see chart 4(b)). In this case, the major part of the mutants are killed by the first test cases of the TS without applying `S2` and consequently, `EMINENT` requires less time to kill the mutants than `OUTRIDER` using a sorted TS.

The strategy that provides the best results in `OUTRIDER` is `S1`, reaching an improvement, with respect to `EMINENT`, of 40% in the overall execution time. This result is obtained using 32 process for testing the CPU-intensive application (see chart 4(e)). Strategy `S3` also provides valuable results, reaching in some scenarios an improvement between 10% - 38% in the overall performance. However, strategy `S4` provides slightly better results than `EMINENT`, it being the best case scenario the testing process of the image filtering application, where 2 equivalent mutants and 19 cloned mutants, divided in 13 domains, have been detected, obtaining a reduction of 20% in the total execution time.

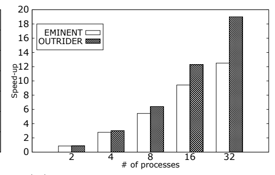
In conclusion, `OUTRIDER` provides better resource usage efficiency than `EMINENT`. These experiments show that strategies `S1` and `S3` clearly provides a significant improvement in the overall system performance. Moreover, in terms of scalability, that is, the performance obtained when the computational resources are increased, these strategies are better than `S2` and `S4`. The main reason of this behaviour is two-fold: First, strategies `S1` and `S3` are less dependent of both the TS and the mutant set, which mainly focus on exploiting the resources of the system. Second, strategies `S2` and `S4` strongly depend of the TS and the mutant set, respectively. Only in those cases where the killer test is not located in the first positions of the TS and the mutant set contains several cloned and equivalent mutants, `OUTRIDER` using `S2` and `S4` executes faster than `EMINENT`.



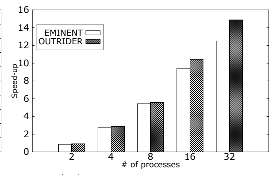
(a) `S1`: Parallelizing TS



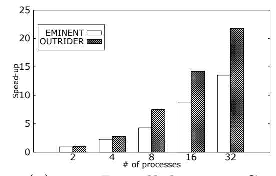
(b) `S2`: Sorting TS



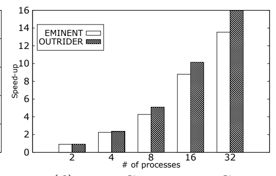
(c) `S3`: Distributing TS



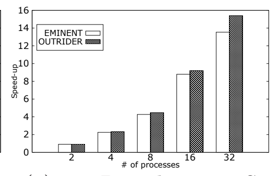
(d) `S4`: Grouping



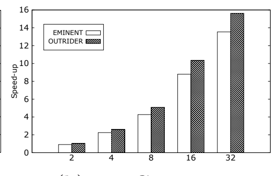
(e) `S1`: Parallelizing TS



(f) `S2`: Sorting TS



(g) `S3`: Distributing TS



(h) `S4`: Grouping

Figure 4: Performance of the testing process using `EMINENT` and `OUTRIDER` with a single strategy

4.2 Performance evaluation using different strategies in `OUTRIDER`

In this section we analyze the performance of `OUTRIDER` when different strategies are used. For the sake of clarity, we only show those configurations that obtain the most representative results, which are depicted in Table 3.

	Configuration							
Strategy	C_{12}	C_{13}	C_{23}	C_{123}	C_{34}	C_{134}	C_{234}	C_{1234}
`S1`: Parallelizing TS	✓	✓		✓		✓		✓
`S2`: Sorting TS	✓		✓	✓			✓	✓
`S3`: Distribution		✓	✓	✓	✓	✓	✓	✓
`S4`: Grouping					✓	✓	✓	✓

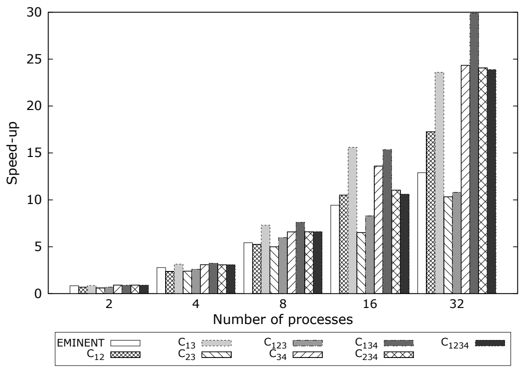
Table 3: Configuration of the strategies used in `OUTRIDER`

Figure 5 shows the results of executing the testing process with `EMINENT` and `OUTRIDER` using different configurations. In these charts, X-axis shows the number of processes used and Y-axis shows the obtained speed-up with respect to a sequential execution.

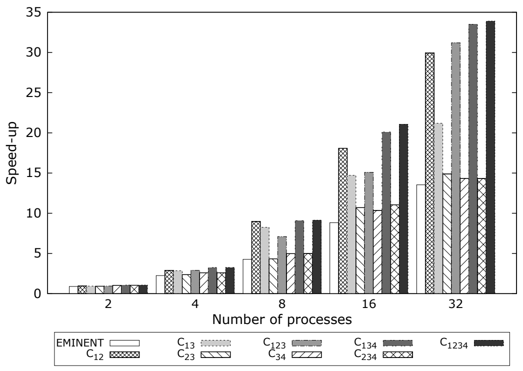
Figure 5(a) presents the results of the testing process using the image filtering application. In general, `OUTRIDER` achieves better performance than `EMINENT`. Both approaches provide similar performance when 2 and 4 processes are used. However, when the number of computational resources increases, the difference of performance between `EMINENT` and `OUTRIDER` increases as well. For instance, when 32 processes are used, `OUTRIDER` with $C_{13}, C_{34}, C_{134}, C_{1234}$ obtains a reduction in the total execution time, with respect to `EMINENT`, of 50%, 50.5%, 60% and 50%, respectively. Configuration C_{134} is particularly relevant. In this case, `OUTRIDER` achieves a speed-up of 2.3 with respect to `EMINENT`. This improvement in the overall testing performance is mainly reached because of the `S4` strategy, which accelerates the MT process by avoiding the complete execution of some mutants (see Section 3.4). However, these configurations using the strategy `S2` do not guarantee an improvement in the total execution time. For instance, `OUTRIDER` using C_{23} executes 19% slower than `EMINENT`.

Figure 5(b) shows the results of the testing process using the CPU-intensive application. Similarly to the previous experiments, we obtain a similar tendency in the system scalability. That is, using few computational resources provides almost the same performance for both approaches. However, increasing the number of computational resources provides a proportional improvement in the overall system performance. In these experiments, all the configurations used in `OUTRIDER` provide a better performance than `EMINENT`. In particular, $C_{12}, C_{123}, C_{134}, C_{1234}$ are especially relevant because `OUTRIDER` using these configurations executes 62%, 67%, 70% and 71% faster than `EMINENT`, respectively.

In general, the overall system performance is increased when combining different strategies. Configuration C_{134} achieves a significantly better speed-up than `EMINENT` for checking the image filtering application, especially when 32 processors are used. However, configuration C_{1234} only achieves slightly better results than C_{134} when 8 and 16 processors are used to check the CPU-intensive application. It is important to remark that the best results are obtained when different strategies are combined using `OUTRIDER`, especially strategies `S1` and `S4`. In conclusion, `OUTRIDER` provides a better resource usage efficiency than `EMINENT`, which is reflected in the scalability obtained, reaching in some cases a speed-up higher than the number of CPU cores used.



(a) Image filter application



(b) Computational intensive application

Figure 5: Performance evaluation of `EMINENT` and `OUTRIDER` using different strategies

5 Conclusions

In this paper we have presented `OUTRIDER`, an HPC-based optimization for the MT process in HPC systems. This optimization consists of 4 strategies aimed to improve the resource usage efficiency, which uses `EMINENT` as basis. Also, an experimental phase has been carried out to evaluates the effectiveness and scalability of `OUTRIDER`.

The experimental study carried out in this work shows that `OUTRIDER` outperforms previous proposals to improve performance of the MT process. In general, `OUTRIDER` provides the best results when different strategies are combined, specially `S1`, `S3` and `S4`, obtaining in some scenarios an improvement of 70% in the overall performance with respect to `EMINENT`. On the contrary, the results obtained when `S2` is used shows that an improvement in the overall performance is not guaranteed. For instance, there are scenarios where `OUTRIDER` executes 66% faster than `EMINENT` (see C_{12} and C_{123} in Figure 5(b)) and there is other scenarios where `OUTRIDER` executes 20% slower than `EMINENT` (see C_{23} in Figure 5(a)).

As future work, we will evaluate the possibility to include some mechanisms for automatically selecting these strategies to be applied in a given MT environment.

References

[1] Sergey Bastrakov, Iosif Meyerov, Victor P. Gergel, Arkady Gonoskov, Anton V. Gorshkov, Evgeny Efimenko, M. Ivanchenko, Mikhail Yu. Kirillin, A. Malova, G. Osipov, V. Petrov, Igor Surmin, and A. Vildemanov. High performance computing in biomedical applications. In *International Conference on Computational Science*, pages 10–19, 2013.

[2] P.C. Cañizares, M.G. Merayo, and A. Núñez. Eminent: Embarrassingly parallel mutation testing. In *International Conference on Computational Science*, volume 80, pages 63–73. Elsevier, 2016.

[3] B. Choi and A.P. Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20(2):135–152, 1993.

[4] L. Deng, A.J. Offutt, P. Ammann, and N. Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 2016.

[5] R. Gopinath, C. Jensen, and A. Groce. Topsy-turvy: a smarter and faster parallelization of mutation analysis. In *International Conference on Software Engineering Companion*, pages 740–743. ACM, 2016.

[6] W. Grosso. *Java RMI*. O'Reilly & Associates, Inc., 1st edition, 2001.

[7] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Practice and Research Techniques*, pages 94–98. IEEE, 2008.

[8] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *International Working Conference on Source Code Analysis and Manipulation*, pages 147–156. IEEE, 2016.

[9] E.W. Krauser, A.P. Mathur, and Vernon J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[10] F. Li, J. Yu, Z. Cao, J. Zhang, M. Chen, and X. Li. Experimental demonstration of four-channel wdm 560 gbit/s 128qam-dmt using im/dd for 2-km optical interconnect. *Journal of Lightwave Technology*, 2016.

[11] Y. Ma and S. Kim. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software*, 115:18–30, 2016.

[12] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 supercomputer sites, 2016. `http://www.top500.org`.

[13] R.M. Hierons M.G., Merayo, and M. Núñez. Controllability through nondeterminism in distributed testing. In *International Conference on Testing Software and Systems*, pages 89–105, 2016.

[14] A.J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability*, 7(3):165–192, 1997.

[15] M. Papadakis, Y. Jia, M. Harman, and Y. Le-Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *International Conference on Software Engineering*, volume 1, pages 936–946. IEEE, 2015.

[16] D.R. Penas, P. González, J.A. Egea, J. R. Banga, and R. Doallo. Parallel metaheuristics in computational biology: An asynchronous cooperative enhanced scatter search method. In *International Conference on Computational Science*, pages 630–639, 2015.

[17] K. Qureshi and H. Rashid. A performance evaluation of rpc, java rmi, mpi and pvm. *Malaysian Journal of Computer Science*, 18(2):38–44, 2005.

[18] P. Reales and M. Polo. Bacterio: Java mutation testing tool: A framework to evaluate quality of tests cases. In *International Conference on Software Maintenance*, pages 646–649. IEEE, 2012.

[19] P. Reales and M. Polo. Parallel mutation testing. *Journal of Software Testing, Verification and Reliability*, 23(4):315–350, 2013.

[20] I. Saleh and K. Nagi. Hadoopmutator: A cloud-based mutation testing framework. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 172–187. Springer, 2014.

[21] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *International Conference of Software Engineering Conference*, pages 297–298. ACM, 2009.

[22] E. Wong. *On mutation and data flow*. PhD thesis, Purdue University, 1993.

[23] F. Wu, M. Harman, Y. Jia, and J. Krinke. Homi: Searching higher order mutants for software improvement. In *International Symposium on Search Based Software Engineering*, pages 18–33. Springer, 2016.